Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2018

Lecture 16

Ana Bove

May 21st 2018

Recap: PDA, TM

- Push-down automata;
- Brief on undecidable problems;
- Brief on models of computation;
- Turing machines:
  - Defined by a 6-tuple $(Q, \Sigma, \delta, q_0, \Box, F)$;
  - Has an infinite tape on both sides;
  - Has a head that reads/writes and moves to left or right;
  - Partial transition function $\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathsf{L}, \mathsf{R}\}$ with $\Sigma' = \Sigma \cup \Box$;
  - Language accepted by a TM;
  - Turing decider;
  - Recursive and recursively enumerable languages.

Overview of Today's Lecture

- More on Turing machines;
- Summary of the course.

Contributes to the following learning outcome:

- Explain and manipulate the diff. concepts in automata theory and formal lang;
- Determine if a certain word belongs to a language;
- Define Turing machines performing simple tasks;
- Differentiate and manipulate formal descriptions of lang, automata and grammars.

Example of a Turing Decider

The following TM accepts the language $\mathcal{L} = \{ww^r \mid w \in \{0,1\}^*\}$.

Let $\Sigma = \{0, 1, X, Y\}$, $Q = \{q_0, \ldots, q_7\}$ and $F = \{q_7\}$,

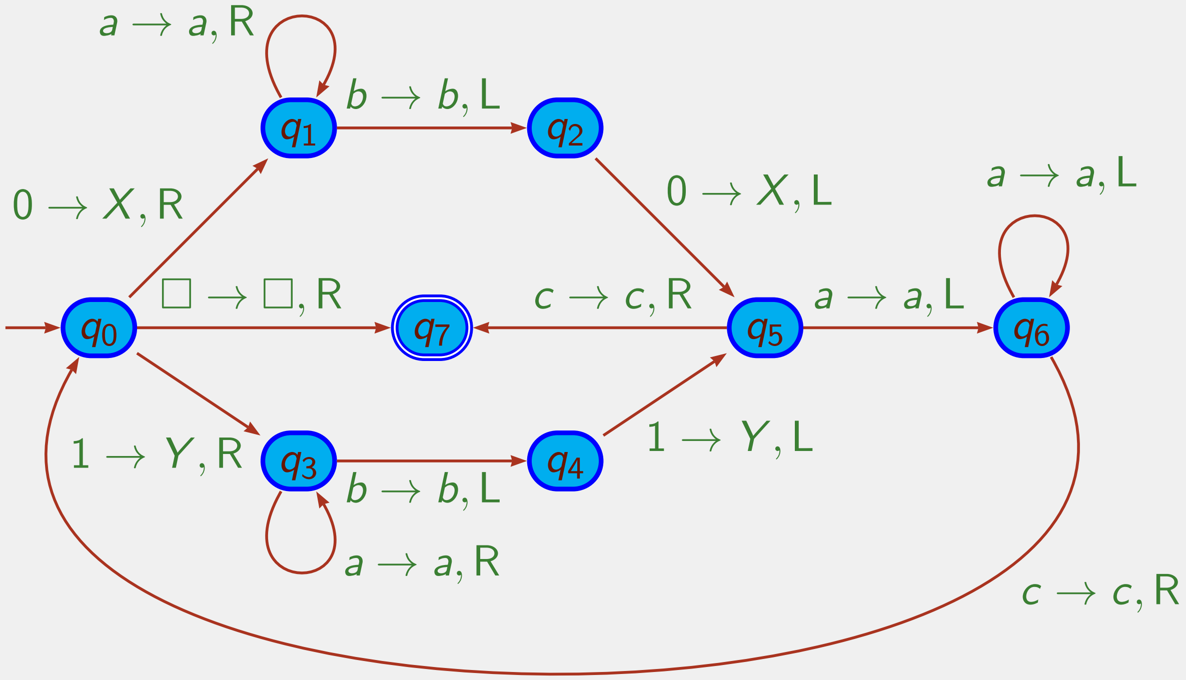
Let $a \in \{0,1\}$, $b \in \{X, Y, \square\}$, and $c \in \{X, Y\}$.

$$\begin{array}{lll}
\delta(q_0, 0) = (q_1, X, \mathrm{R}) & \delta(q_0, 1) = (q_3, Y, \mathrm{R}) & \delta(q_0, \square) = (q_7, \square, \mathrm{R}) \\
\delta(q_1, a) = (q_1, a, \mathrm{R}) & \delta(q_3, a) = (q_3, a, \mathrm{R}) & \\
\delta(q_1, b) = (q_2, b, \mathrm{L}) & \delta(q_3, b) = (q_4, b, \mathrm{L}) & \\
\delta(q_2, 0) = (q_5, X, \mathrm{L}) & \delta(q_4, 1) = (q_5, Y, \mathrm{L}) & \\
\delta(q_5, a) = (q_6, a, \mathrm{L}) & & \delta(q_5, c) = (q_7, c, \mathrm{R}) \\
\delta(q_6, a) = (q_6, a, \mathrm{L}) & \delta(q_6, c) = (q_0, c, \mathrm{R}) &
\end{array}$$

How easy is to understand the "program"?

Transition Diagram of a TM for $\{ww^r \mid w \in \{0,1\}^*\}$

Let $a \in \{0,1\}$, $b \in \{X, Y, \Box\}$, and $c \in \{X, Y\}$.



High-level Description of a TM for $\{ww^r \mid w \in \{0,1\}^*\}$

1. If in the initial state q_0 we read $\Box$, the word is empty so we move to q_7 and accept. Otherwise, if we read 0 (resp. 1) then we mark it with X (resp. Y), move R and change to the state q_1 (resp. q_3) that will search for the corresponding 0 (resp. 1) at the end of the input.

2. When q_1 (resp. q_3) is searching for the corresponding 0 (resp. 1) we move R over 0's and 1's.
   If we read X, Y or $\Box$ then we have found the end of the unchecked input, so we move L to the first unmarked symbol, and change to the state q_2 (resp. q_4) that will check if the symbol is indeed a 0 (resp. 1).
   Otherwise, we halt.

3. If in q_2 (resp. q_4) we indeed read a 0 (resp. 1) then the input is still correct, so we mark the 0 (resp. 1) with X (resp. Y), move L and change to state q_5 which will check if we are done or otherwise will go back to the first unchecked symbol in the left.

High-level Description of a TM for $\{ww^r \mid w \in \{0,1\}^*\}$ (Cont.)

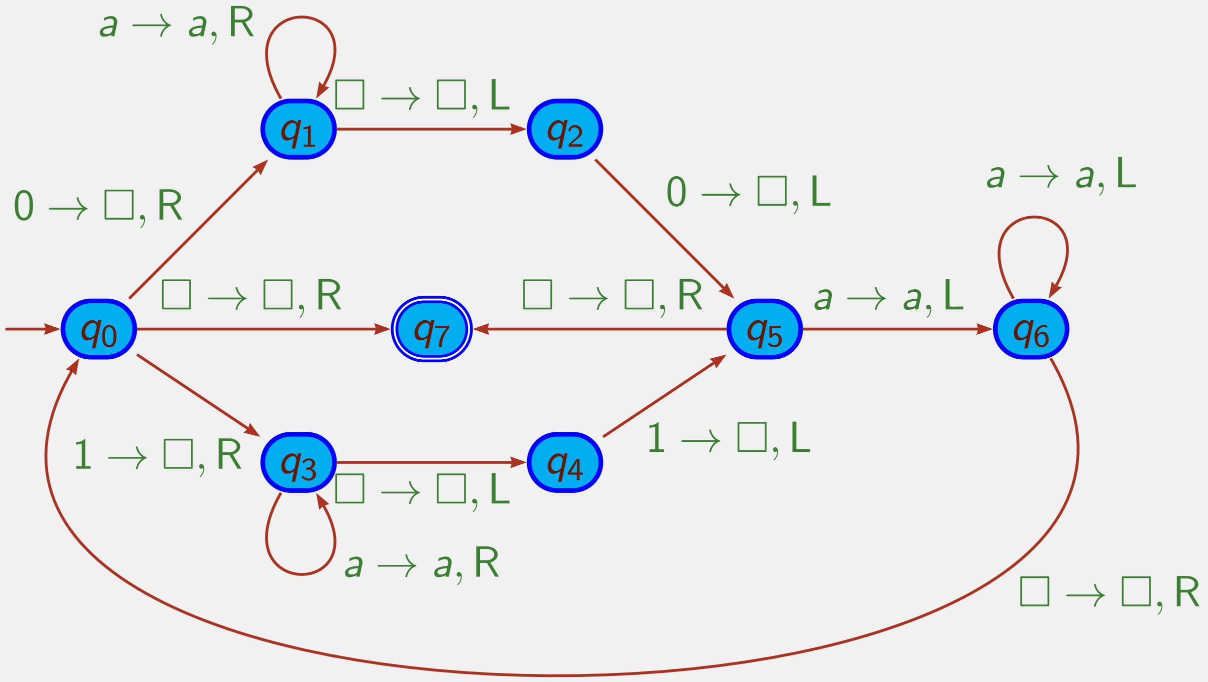
4. If in q_5 we read X or Y, then we have checked that the whole word is of the form ww^r, so we move to q_7 and accept.
   If we instead read 0 or 1 then there are still symbols to check, so we move L and change to the state q_6 that will move to the first unchecked symbol.

5. In q_6 we move L over 0's and 1's to reach the first unchecked symbol in the left.
   If while moving left we read X or Y then we have passed all unchecked symbols, so we move R and change to the initial state q_0 to repeat the procedure with the rest of the (unchecked) input.

A Different TM for $\{ww^r \mid w \in \{0,1\}^*\}$

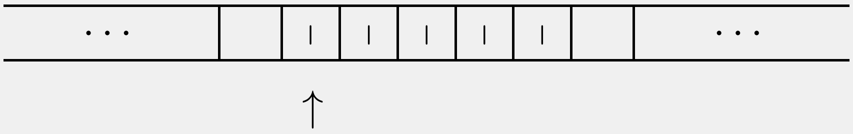
This TM will destroy the input when checking for the answer.

Let $a \in \{0,1\}$.

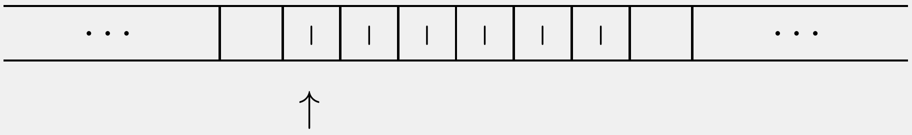


Coding the Natural Numbers

Unary Coding The number 0 is represented by the empty symbol □ and a number $n \neq 0$ is represented with n consecutive symbols.
The number 5 is then represented as



Kleene's Coding The natural number n is represented with $n + 1$ consecutive symbols.
The number 5 is then represented as



Examples

How can we write a TM that compute the following functions over the Natural numbers:

1. Successor and predecessor;
2. Addition and subtraction;
3. Multiplication.

Closure Properties

Recursive languages are closed under union, intersection, concatenation, closure, complement and difference.

Recursive enumerable languages are closed under union, intersection, concatenation and closure but not complement and difference.

Church-Turing Thesis (AKA Church Thesis)

A function is *algorithmically computable* if and only if it can be defined as a Turing Machine.

(Recall that the λ-calculus and Turing machines were shown to be computationally equivalent).

Note: This is not a theorem and it can never be one since there is no precise way to define what it means to be *algorithmically computable*.

However, it is strongly believed that both statements are true since they have not been refuted in the ca. 80 years which have passed since they were first formulated.

Turing Completeness

Definition: A collection of data-manipulation rules (for example, a programming language) is said to be *Turing complete* if and only if such system can simulate any single-taped Turing machine.

Example: Recursive functions and λ-calculus.

The three models of computation were shown to be equivalent by Church, Kleene & (John Barkley) Rosser (1934–6) and Turing (1936-7).

Variants of Turing Machines

What follows are some variants, extensions and restrictions to the notion of TM that we presented, none of them modifying the power of the TM.

- Storage in the state;
- Multiple tracks in one tape;
- Subroutines;
- Multiple tapes;
- Non-deterministic TM;
- Semi-infinite tapes.

FA vs. PDA vs. TMs

FA: Bounded input and finite set of states;
Can only read and move to the right;
After reading the word it decides whether to accept or not.

PDA: Bounded input and finite set of states;
Can only read and move to the right;
Stack with unbounded memory and LIFO access (last in-first out)
After reading the word it decides whether to accept or not.

TM: Unbound input and finite set of states;
Can read and write, and move left and right;
Unrestricted access in unbounded memory;
If the TM is in a final state when when it halts, then the input is accepted.

Differences between FA vs. TMs

- TMs can read and write the tape/input;
- TMs have an infinite tape (on one or both directions);
- TMs can move to the right and to the left on the tape/input;
- TMs have a special states for accepting (and eventually even rejecting) independent of the content of the tape;
- TMs can loop or get stack.

Overview of the Course

We have covered chapters 1–5 + 7 + (8):

Formal proofs: mainly proofs by induction;

Regular languages: DFA, NFA, ϵ-NFA, RE;
Algorithms to transform one formalism to the other;
Pumping lemma for RL;
Closure and decision properties of RL;

Context-free languages: CFG;
Pumping lemma for CFL;
Closure and decision properties of CFL;

Turing machines: Just the idea.

Formal Proofs

We have used formal proofs along the course to prove our results.

Mainly proofs by induction:

- By induction on the structure of the input argument;
- By induction on the length of the input string;
- By induction on the length of the derivation;
- By induction on the height of a parse tree.

Finite Automata and Regular Expressions

FA and RE can be used to model and understand a certain situation/problem.

Example: Consider the problem with the man, the wolf, the goat and the cabbage.

Also the Gilbreath’s principle. There we went from NFA → DFA → RE.

They can also be used to describe (parts of) a certain language.

Example: RE are used to specify and document the lexical analyser (*lexer*) in languages (the part of the compiler reading the input and producing the different *tokens*).

The implementation performs the steps RE → NFA → DFA → min DFA.

Example: Using Regular Expression to Identify the Tokens

```
Tokens = Space (Token Space)*
Token  = TInt | TId | TKey | TSpec
TInt   = Digit Digit*
Digit  = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ |
         ’7’ | ’8’ | ’9’
TId    = Letter IdChar*
Letter = ’A’ | ... | ’Z’ | ’a’ | ... | ’z’
IdChar = Letter | Digit
TKey   = ’i’’f’ | ’e’’l’’s’’e’ | ...
TSpec  = ’+’’+’ | ’+’ | ...
Space  = (’ ’ | ’\n’ | ’\t’)*
```

Regular Languages

Intuitively, a language is regular when a machine needs only limited amount of memory to recognise it.

We can use the Pumping lemma for RL to show that a certain language is not regular.

We can use closure properties for RL to show that a certain language is or is not regular.

There are many decision properties we can answer for RL.
Some of them are:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}? \qquad \mathcal{L} = \mathcal{L}'?$$

Context-free Grammars

CFG play an important role in the description and design of programming languages and compilers.

CFG are used to define the syntax of most programming languages.

Parse trees reflect the structure of the word.

In a compiler, the parser takes the input into its abstract syntax tree (which also reflects the structure of the word but abstracts from some concrete features).

A grammar is ambiguous if a word in the language has more than one parse tree.

Context-free Languages

These languages are generated by CFG.

It is enough to provide a stack to a ϵ-NFA in order to recognise these languages.

We can use the Pumping lemma for CFL to show that a certain language is not context-free.

There are only a few decision properties we can answer for CFL. Mainly:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}?$$

However there are no algorithms to determine whether $\mathcal{L} = \mathcal{L}'$.

There is no algorithm either to decide if a grammar is ambiguous or a language is inherently ambiguous.

Turing Machines

Simple but powerful devices.

They can be thought of as a DFA plus a tape which we can read and write, and that we can access randomly.

Define the recursively enumerated languages.

It allows the study of *decidability*: what can or cannot be done by a computer (halting problem).

Computability vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

Learning Outcome of the Course

- Explain and manipulate the different concepts in automata theory and formal languages;
- Have a clear understanding about the equivalence between (non-)deterministic finite automata and regular expressions;
- Acquire a good understanding of the power and the limitations of regular languages and context-free languages;
- Prove properties of languages, grammars and automata with rigorously formal mathematical methods;
- Design automata, regular expressions and context-free grammars accepting or generating a certain language;
- Describe the language accepted by an automata or generated by a regular expression or a context-free grammar;
- Simplify automata and context-free grammars;
- Determine if a certain word belongs to a language;
- Define Turing machines performing simple tasks;
- Differentiate and manipulate formal descriptions of languages, automata and grammars.

Overview of Next Lecture

- Old exams.
  Please try to do some of the exercises before!